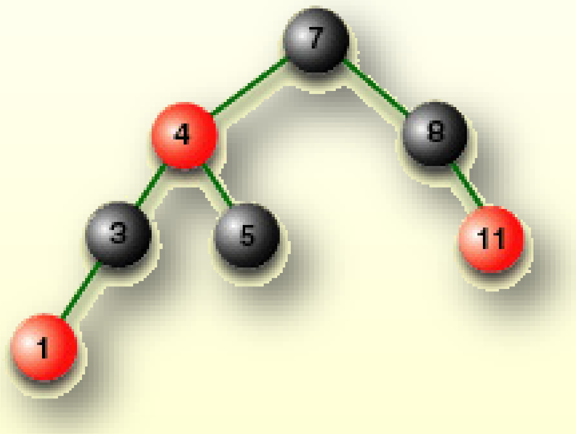
CS161: Design and Analysis of Algorithms



Lecture 7

Leonidas Guibas

Outline

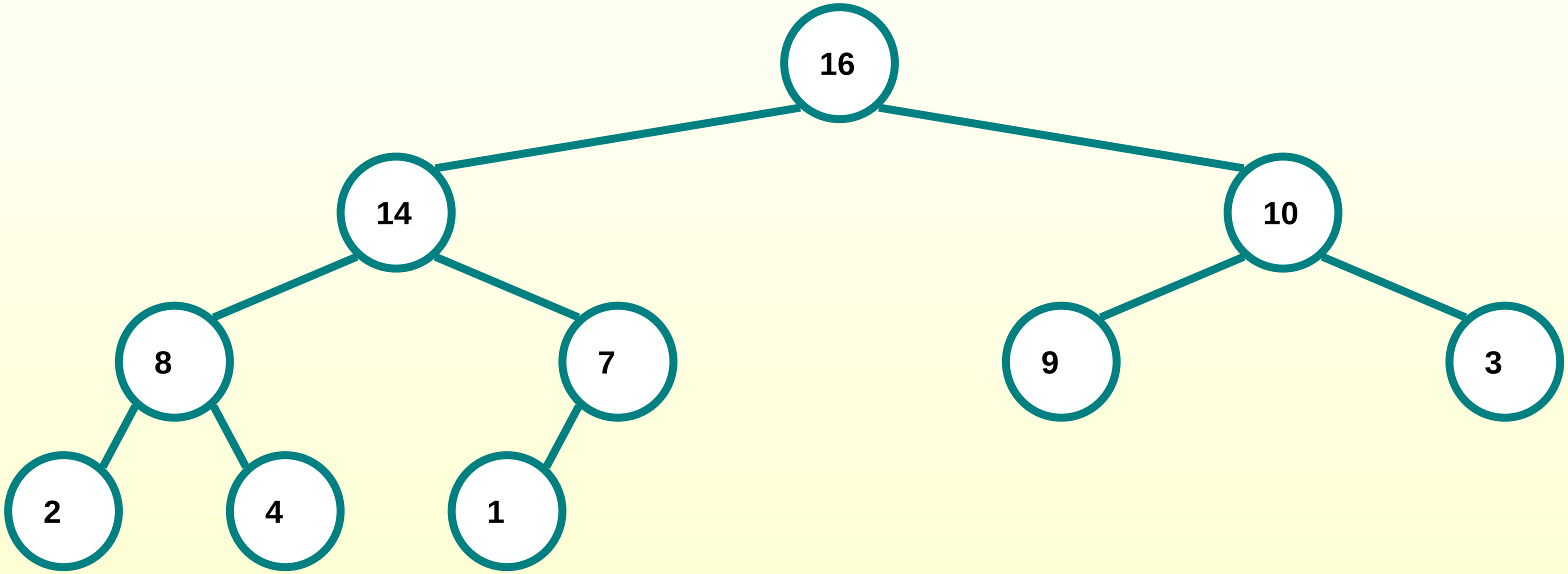
- Review of last lecture: Heaps and HeapSort
- All sorts we saw take $\Omega(n \log n)$ time
- Sorting in linear time
  - Lower bounds for sorting
  - CountingSort
  - RadixSort
  - BucketSort

Slides modified from

- http://www.cs.unc.edu/~plaisted/comp122/00-intro.ppt
- http://school.eecs.wsu.edu/undergraduate/cpts/courses/223

Heaps

- A *heap* can be seen as a complete binary tree



- A tree in which every node holds a key larger than or equal to those of its children

Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property
  - Action: let the value of the parent node “sift down” so subtree at *i* satisfies the heap property
    - Fix up the relationship between i, l, and r recursively

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Abstract Data Structure: Priority Queue

- Insert(S, x) inserts the element x into set S
- Maximum(S) returns the element of S with the maximum key
- ExtractMax(S) removes and returns the element of S with the maximum key
- ChangeKey(S, i, key) changes the key for element i to something else

- All these operations can be implemented in O(lg n) time using a heap

Comparison Sorting

Sort	Worst Case	Average Case	Best Case	Comments
InsertionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N)$	Fast for small N
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Requires memory
HeapSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Large constants
QuickSort	$\Theta(N^2)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Small constants

Lower Bound on Sorting

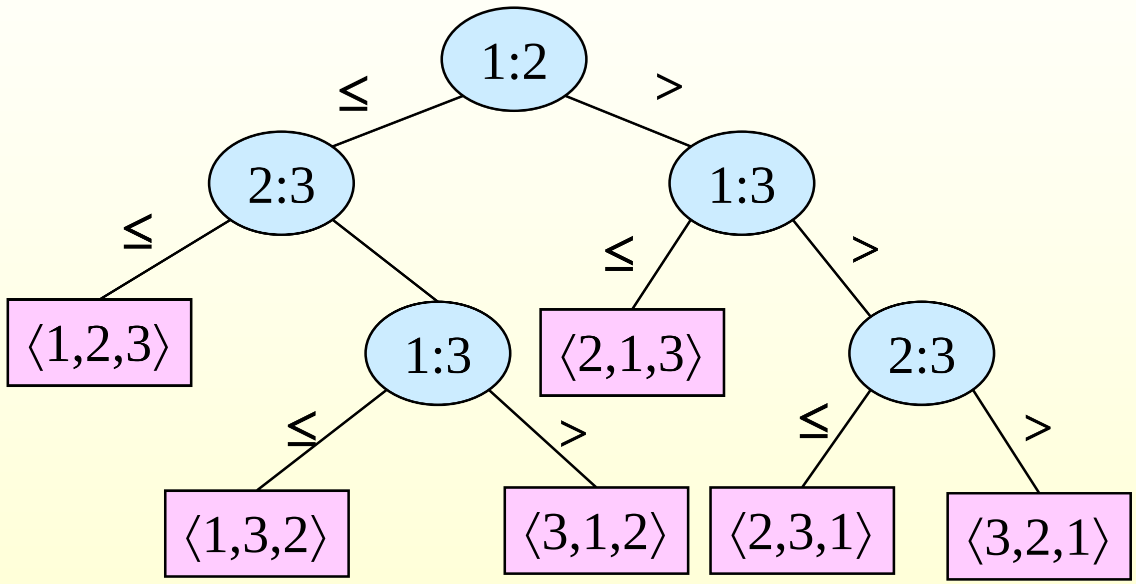
What is the best we can do on comparison based sorting?

- Best worst-case sorting algorithm (so far) is O(N log N)
  - Can we do better?
- Can we prove a lower bound on the sorting problem, independent of the algorithm?
  - For comparison sorting, no, we cannot do better than O(N log N)
  - Can show lower bound of Ω(N log N)

A lower bound is something that applies to a whole class of algorithms, not just a single algorithm

Decision Tree Approach

For InsertionSort operating on three elements.



Simply unroll all loops for all possible inputs.

Node *i*:*j* means compare A[*i*] to A[*j*].

Leaves show outputs;

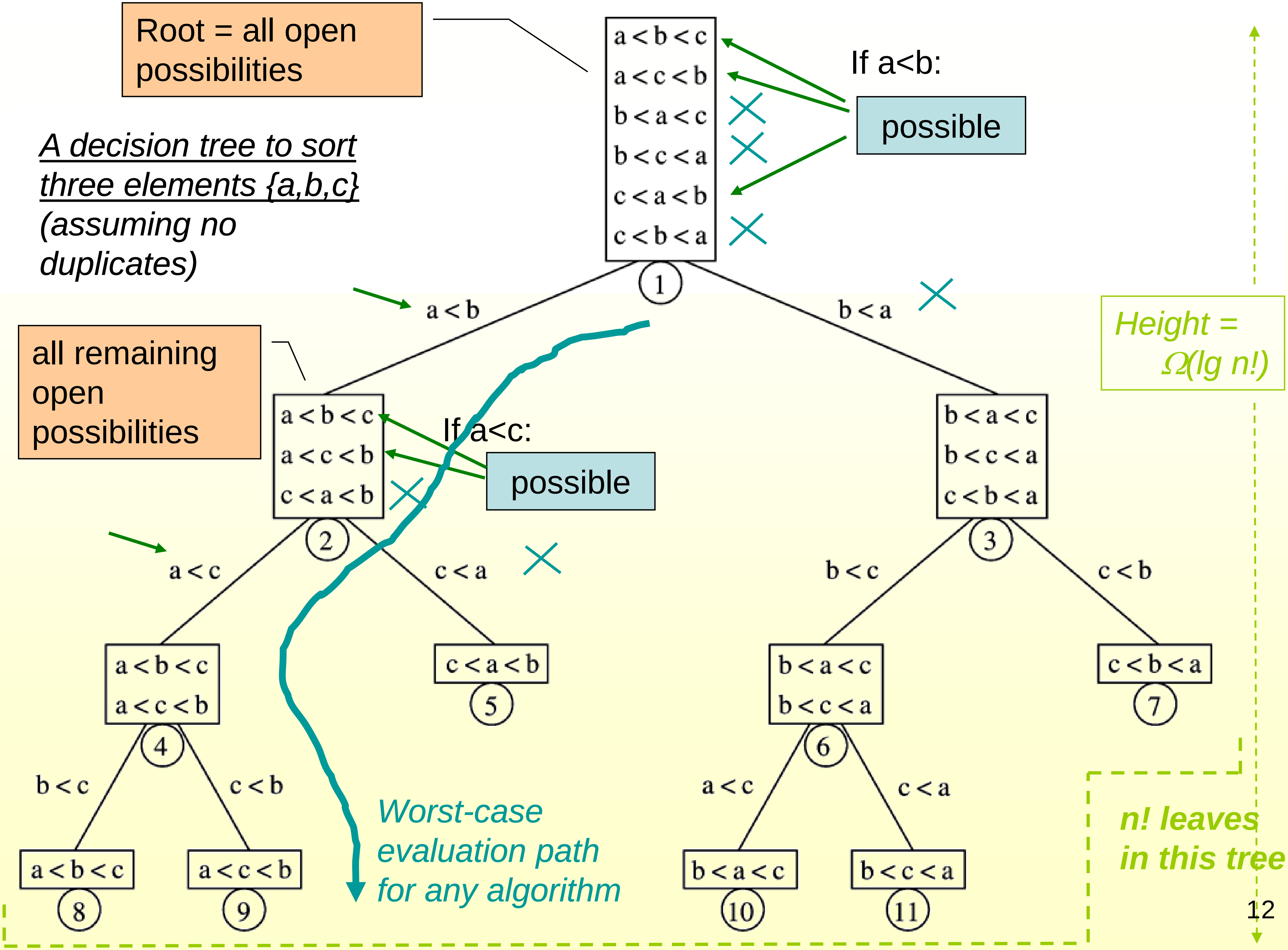
No two paths go to same leaf!

Contains 3! = 6 leaves.

Comparison-Based Decision Trees for Sorting

A comparative *decision tree* is a binary tree where:

- Each internal node is a comparison
  - It implicitly holds all remaining undecided possibilities (for future decisions)
- The path to each node
  - represents an already determined sorted prefix of elements (partial sort info)
- Each branch
  - represents an outcome of a particular comparison
- Each leaf
  - represents a particular final ordering of the original array elements (everything is decided)


Root = all open possibilities
A decision tree to sort three elements {a,b,c} (assuming no duplicates)
a<b<c
a<c<b
b<a<c
b<c<a
c<a<b
c<b<a
If a<b:
possible
1
a<b
b<a
Height = Ω(lg n!)
all remaining open possibilities
a<b<c
a<c<b
c<a<b
If a<c:
possible
2
b<a<c
b<c<a
c<b<a
3
a<c
c<a
b<c
c<b
a<b<c
a<c<b
4
c<a<b
5
b<a<c
b<c<a
6
c<b<a
7
b<c
c<b
a<c
c<a
a<b<c
8
a<c<b
9
b<a<c
10
b<c<a
11
Worst-case evaluation path for any algorithm
n! leaves in this tree

Decision Tree

- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \leq a_j$ is made.
  - If $a_i \leq a_j$, follow left subtree, else follow right subtree.
- When we come to a leaf, a full ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - Hence, each of the n! permutations must appear at one or more of the leaves of the decision tree.

Decision Trees for Sorting

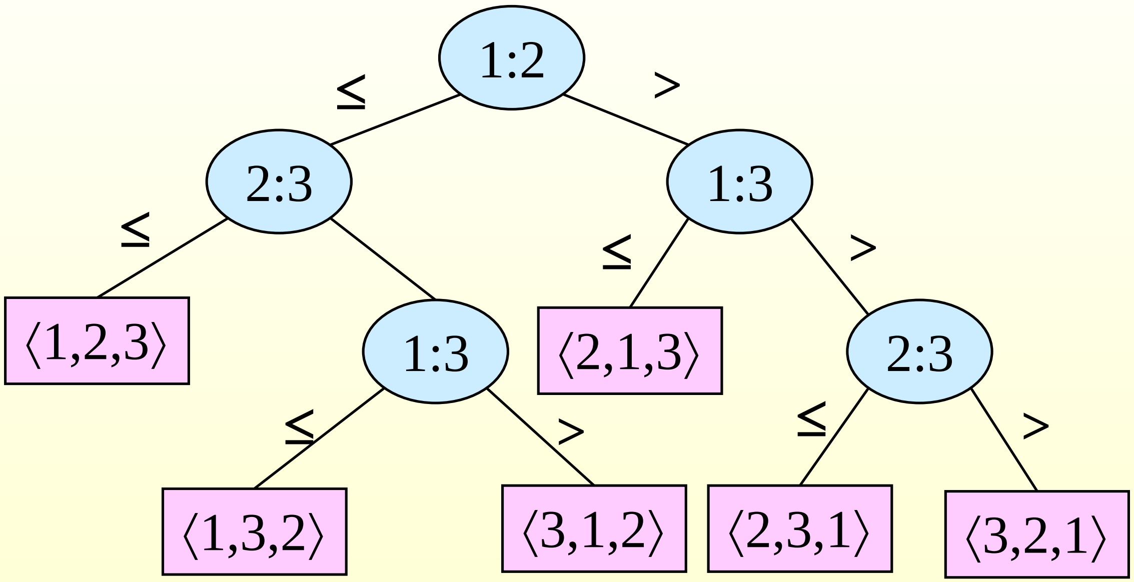
- The logic of *any sorting algorithm* that uses comparisons can be represented by a decision tree
- In the worst case, the number of comparisons used by the algorithm equals the height of the decision tree
- In the average case, the number of comparisons is the mean depth of all leaves
- There are N! different orderings of N elements

A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is the
  - length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - a lower bound on the height of all decision trees in which each permutation appears as a reachable leaf.

Optimal Sort of Three Elements

Any sort of six elements has 5 internal nodes.



There must be a worst-case path of length ≥ 3.

Lower Bound for Comparison Sorting

Lemma: *A binary tree with L leaves must have depth at least* $\lceil \lg L \rceil$

Any sorting decision tree has N! leaves

Theorem: *Any comparison sort must require at least* $\lceil \lg N! \rceil = \Theta(N \lg N)$ *comparisons in the worst case*

Lower Bound for Comparison Sorting

<u>Theorem:</u> *Any comparison sort requires Ω(N log N) comparisons*

- Proof (using Stirling's approximation)

$$N! = \sqrt{2\pi N}(N/e)^N(1+\Theta(1/N))$$

$$N! > (N/e)^N$$

$$\log(N!) > N\log N - N\log e = \Theta(N\log N)$$

$$\therefore \log(N!) = \Omega(N\log N)$$

Implications of Lower Bound

- Comparison-based sorting cannot be achieved in less than $\Omega(n \lg n)$ steps

=> MergeSort, HeapSort are optimal worst-case asymptotically optimal

=> QuickSort is not optimal, but very efficient in practice

=> InsertionSort, is sub-optimal, even in practice

Non-Comparative Sorts

- Counting sort
- Radix sort
- Bucket sort

Integer Sorting

- Some input properties allow to eliminate the need for comparison
  - Because we can decide the order of the keys some other way
    - E.g., sorting an employee database by age of employees
- Counting Sort (for small integer data)
  - *Given array $A[1..N]$, where $1 \le A[i] \le M$*
  - Create array C of size M, where C[i] is the number of i's in A
  - Use C to place elements into new sorted array B
  - Running time $\Theta(N+M) = \Theta(N)$ if $M = \Theta(N)$

Counting Sort: Example

Input A:

1	2	3	4	5	6	7	8	9	10
0	3	2	1	3	2	1	2	2	3

N=10
M=4

(all elements in input between 0 and 3)

Count array C:

0	**1**
1	**2**
2	**4**
3	**3**

Output sorted array B:

1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	2	2	3	3	3

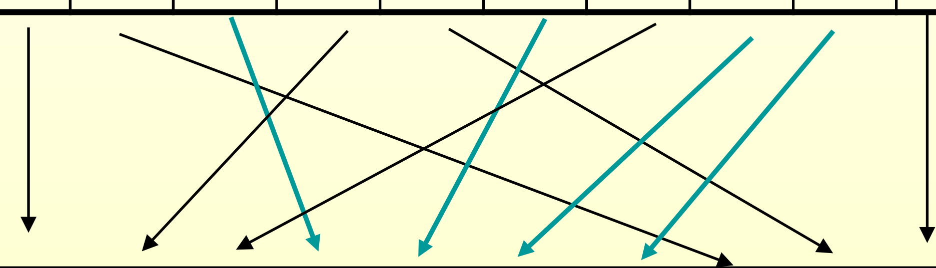
Time = O(N + M)

If (M < N), Time = O(N)

Stable vs. Nonstable Sorting

- A stable sorting method is one which preserves the original input order among duplicates in the output

Input:	0	3	2	1	3	2	1	2	2	3



Output:	0	1	1	2	2	2	2	3	3	3

Useful when each data is a struct of form {key, value} and we care to preserve the ordering of the values

Making Counting Sort Stable

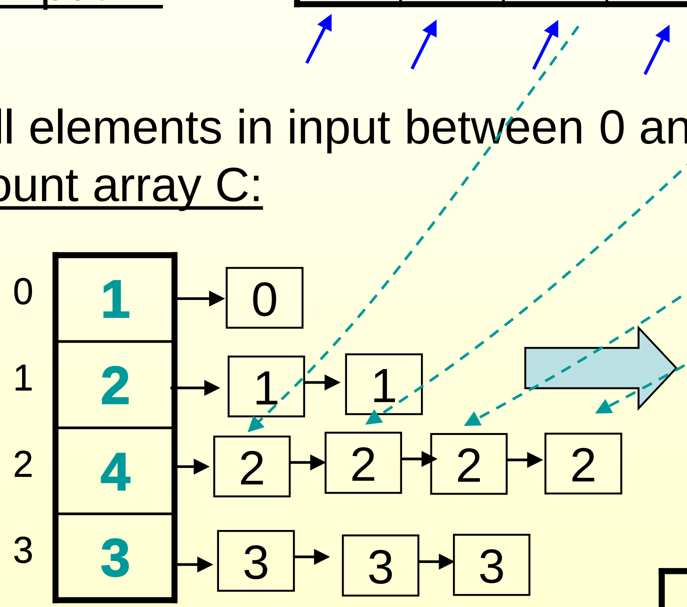
Input A:

1	2	3	4	5	6	7	8	9	10
0	3	2	1	3	2	1	2	2	3

N=10
M=4

(all elements in input between 0 and 3)

Count array C:

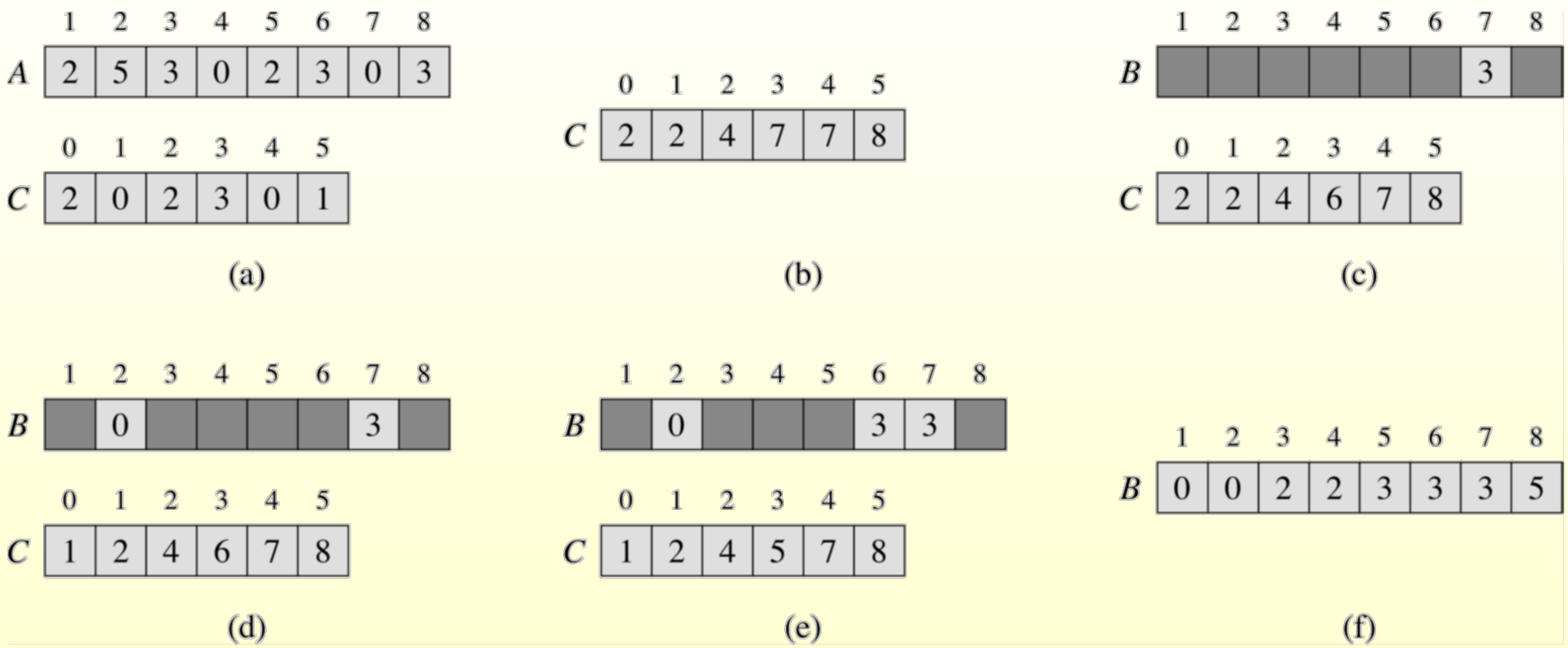


Output sorted array:

1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	2	2	3	3	3

But this algorithm has too much overhead

Example



1 2 3 4 5 6 7 8
A 2 5 3 0 2 3 0 3
0 1 2 3 4 5
C 2 0 2 3 0 1
(a)
0 1 2 3 4 5
C 2 2 4 7 7 8
(b)
1 2 3 4 5 6 7 8
B 3
0 1 2 3 4 5
C 2 2 4 6 7 8
(c)
1 2 3 4 5 6 7 8
B 0 3
0 1 2 3 4 5
C 1 2 4 6 7 8
(d)
1 2 3 4 5 6 7 8
B 0 3 3
0 1 2 3 4 5
C 1 2 4 5 7 8
(e)
1 2 3 4 5 6 7 8
B 0 0 2 2 3 3 3 5
(f)

Stable Counting Sort

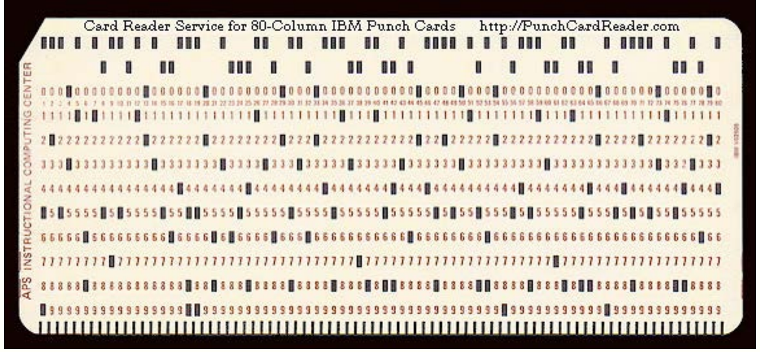
```
CountingSort(A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Stable Counting Sort Summary

- Counting sort:
  - Assumption: input is n numbers in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element i
    - Use that number to place i in position k of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place:
    - O(n) array to hold sorted output
    - O(k) array for scratch storage (the counts)

From A Different Era

Radix Sort



- *How did IBM made its money originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - A card has 72 columns
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix Sort

- Same problem in sorting ordinary decimal numbers
- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort on the *least* significant digit first, use a stable sort

```
RadixSort(A, d)
   for i=1 to d
      StableSort(A) on digit I
```

Example

Radix sort is the algorithm used by card-sorting machines that today may be found only in museums.

input	**output**
32**9**	
45**7**	
65**7**	
83**9**	
43**6**	
72**0**	
35**5**	

Example

Radix sort is the algorithm used by card-sorting machines that today may be found only in museums.

input		output
32**9**	7**2**0	
45**7**	3**5**5	
65**7**	4**3**6	
83**9**	4**5**7	
43**6**	6**5**7	
72**0**	3**2**9	
35**5**	8**3**9	

Example

Radix sort is the algorithm used by card-sorting machines that today may be found only in museums.

input			output
32**9**	7**2**0	**7**20	
45**7**	3**5**5	**3**29	
65**7**	4**3**6	**4**36	
83**9**	4**5**7	**8**39	
43**6**	6**5**7	**3**55	
72**0**	3**2**9	**4**57	
35**5**	8**3**9	**6**57	

Example

Radix sort is the algorithm used by card-sorting machines that today may be found only in museums.

input			output
32**9**	7**2**0	**7**20	329
45**7**	3**5**5	**3**29	355
65**7**	4**3**6	**4**36	436
83**9**	4**5**7	**8**39	457
43**6**	6**5**7	**3**55	657
72**0**	3**2**9	**4**57	720
35**5**	8**3**9	**6**57	839

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits 1…i-1 are sorted
  - Show that sorting next digit i leaves array correctly sorted for digits 1…i
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is the obvious choice:
  - Sort n numbers on digits that range from 1..k
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, this takes $O(n)$ time

Radix Sort

- Sort N numbers, each with k bits
- E.g, input {4, 1, 0, 10, 5, 6, 1, 8}, 4 bits

4	0100	0100	0100	0000	0000	0
1	0001	0000	0000	1000	0001	1
0	0000	1010	1000	0001	0001	1
10	1010	0110	0001	0001	0100	4
5	0101	1000	0101	1010	0101	5
6	0110	0001	0001	0100	0110	6
1	0001	0101	1010	0101	1000	8
8	1000	0001	0110	0110	1010	10
	lsb			msb		

- Radix sort achieves stable sorting
- To sort each column, use counting sort (O(n))
  => To sort d columns, O(dn) time

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approx lg n = 20 operations per number being sorted
- *So why would we ever use anything but radix sort?*

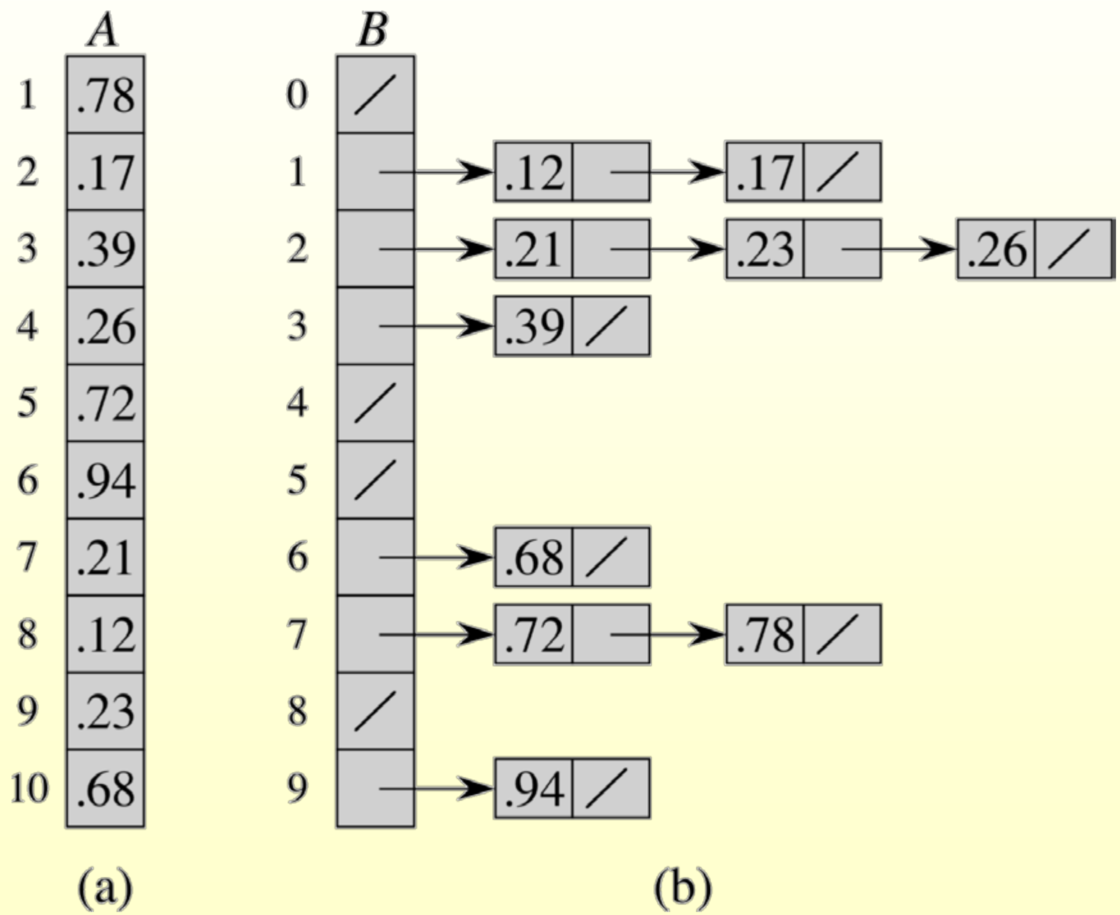
Radix Sort Summary

- Radix sort:
  - Assumption: input has n numbers with d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with the *least* significant first
    - Use a stable sort (like counting sort) for each stage
  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast, stable, and simple
  - Doesn't sort in place

Bucket Sort

- Assume N elements of A uniformly distributed over the range [0,1]
- Create M equal-sized buckets over [0,1], s.t., $M \leq N$
- Add each element of A into appropriate bucket
- Sort each bucket internally
  - Can use recursion here, or
  - Can use something like InsertionSort
- Return concatentation of buckets
- Average case running time $\Theta(N)$
  - assuming each bucket will contain $\Theta(1)$ elements

Bucket Example



BucketSort (A)

Input: $A[1..n]$, where $0 \le A[i] < 1$ for all i.
Auxiliary array: $B[0..n-1]$ of linked lists, each list initially empty.

BucketSort(A)

1. $n \leftarrow length[A]$
2. **for** $i \leftarrow 1$ to n
3. **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. **for** $i \leftarrow 0$ **to** $n - 1$
5. **do** sort list $B[i]$ with insertion sort
6. concatenate the lists $B[i]$s together in order
7. **return** the concatenated lists

\# buckets = # elements

BucketSort: Correctness

Why does it work?

Left as an exercise*.*

(Prove that any two elements land up in the right order regardless of all the others).

Analysis

- Relies on no bucket getting too many values.
- All lines except insertion sorting in line 5 take $O(n)$ altogether.
- Intuitively, if each bucket gets a constant [$O(1)$] number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow$ $O(n)$ sort time for all buckets.
- We "expect" each bucket to have few elements, if elements are evenly distributed.
- But we need to do a careful analysis.

Analysis – Contd.

- RV n_i = no. of elements placed in bucket $B[i]$.
- Insertion sort runs in quadratic time. Hence, time for bucket sort is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \quad \text{(by linearity of expectation)} \qquad (8.1) \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad (E[aX] = aE[X]) \end{aligned}$$

Analysis – Contd.

- **Claim:** $E[n_i^2] = 2 - 1/n$. (8.2)
- **Proof:**
- Define indicator random variables.
  - $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$
  - $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$.
  - $n_i = \sum_{j=1}^{n} X_{ij}$

Analysis – Contd.

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right]$$

$$= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ j\ne k}} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ j\ne k}} E[X_{ij}X_{ik}]\ \text{, by linearity of expectation.} \qquad (8.3)$$

Analysis – Contd.

$$E[X_{ij}^2] = 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}$$

$$= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}$$

$$= \frac{1}{n}$$

$E[X_{ij} X_{ik}]$ for $j \neq k$:

Since $j \neq k$, X_{ij} and X_{ik} are independent random variables.

$$\Rightarrow E[X_{ij} X_{ik}] = E[X_{ij}] E[X_{ik}]$$

$$= \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

Analysis – Contd.

(8.3) is hence,

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n}.$$

Substituting (8.2) in (8.1), we have,

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n)$$

$$= \Theta(n) + O(n)$$

$$= \Theta(n)$$

Non-Comparative Sorts: Summary

- It is possible to sort without comparisons by “looking inside” the keys and exploiting that structure
- Sometimes that’s the only way: sorting strings
- Numbers also have digit representations
  - CountingSort
  - RadixSort
- Or we use digits to sort numbers into buckets
  - BucketSort